Mini project report on

Personal Movie/TV Show Recommendation System

Submitted in partial fulfilment of the requirements for the award of degree of

Bachelor of Technology

in

Computer Science & Engineering

UE23CS351A – DBMS Project

Submitted by:

A Simon Jonathan	**PES2UG24CS801**
Varun Kumar L	**PES2UG24CS827**

under the guidance of

Prof. Gamini Joshi

Assistant Professor

PES University

AUG - DEC 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

FACULTY OF ENGINEERING

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)

Electronic City, Hosur Road, Bengaluru – 560 100, Karnataka, India

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)

Electronic City, Hosur Road, Bengaluru – 560 100, Karnataka, India

CERTIFICATE

This is to certify that the mini project entitled

Personal Movie/TV Show Recommendation System

is a bonafide work carried out by

A Simon Jonathan	**PES2UG24CS801**
Varun Kumar L	**PES2UG24CS827**

In partial fulfilment for the completion of fifth semester DBMS Project (UE23CSS351A) in the Program of Study - Bachelor of Technology in Computer Science and Engineering under rules and regulations of PES University, Bengaluru during the period AUG. 2025 – DEC. 2025. It is certified that all corrections / suggestions indicated for internal assessment have been incorporated in the report. The project has been approved as it satisfies the 5th semester academic requirements in respect of project work.

Signature

Prof. Gamini Joshi

Assistant Professor

DECLARATION

We hereby declare that the DBMS Project entitled **Personal Movie/TV Show Recommendation System** has been carried out by us under the guidance of **Prof. Gamini Joshi, Assistant Professor** and submitted in partial fulfilment of the course requirements for the award of degree of **Bachelor of Technology** in **Computer Science and Engineering** of **PES University, Bengaluru** during the academic semester AUG – DEC 2023.

A Simon Jonathan	**PES2UG24CS801**	[illegible]
Varun Kumar L	**PES2UG24CS827**	Varun Kumar.L

ACKNOWLEDGEMENT

I would like to express my gratitude to **Prof. Gamini Joshi**, Department of Computer Science and Engineering, PES University, for her continuous guidance, assistance, and encouragement throughout the development of this UE23CS351A - DBMS Project.

I take this opportunity to thank **Dr. Sandesh** B J, C, Professor,ChairPerson, Department of Computer Science and Engineering, PES University, for all the knowledge and support I have received from the department.

I am deeply grateful to Prof. Jawahar Doreswamy, Chancellor – PES University, Dr. Suryaprasad J, Vice-Chancellor, PES University for providing to me various opportunities and enlightenment every step of the way. Finally, this DBMS Project could not have been completed without the continual support and encouragement I have received from my family and friends.

ABSTRACT

This project presents a complete movie recommendation platform that integrates a Flask-based backend, a MySQL database, and a dynamic front-end interface. The system allows users to register, log in, browse movies, and submit ratings, which are stored and processed in real time. The recommendation engine is powered by stored procedures and SQL triggers that automatically update movie statistics such as average rating and rating count.

A procedural algorithm generates personalized movie recommendations based on a user's preferred genres and rating history, while fallback logic provides top-rated movies when insufficient data exists.

Multiple nested SQL queries are used to extract genre preferences, compute aggregates, and filter unrated movies to ensure accurate recommendation results. The project demonstrates backend–database integration, automated data consistency via triggers, and the orchestration of server-side logic through MySQL stored procedures. This system highlights scalable design practices suitable for modern content-based recommendation systems.

TABLE OF CONTENTS

INTRODUCTION

The rapid expansion of digital streaming platforms has transformed the way people consume movies and entertainment. With thousands of movies becoming instantly accessible, users often struggle to find content that matches their personal preferences. This challenge has led to the development of intelligent movie recommendation systems, which analyze user behavior and movie attributes to suggest relevant films. At the core of these systems lies a powerful and well-structured database capable of storing, processing, and retrieving complex information efficiently.

A recommendation system depends heavily on how effectively data is captured, organized, and managed. Data such as movie metadata, genres, cast details, user accounts, ratings, reviews, and watch history must be stored in a structured and relational manner. A poorly designed database can lead to slow performance, redundant data, inconsistencies, and inaccurate recommendations — all of which negatively impact the user experience. Therefore, designing a robust SQL-based database becomes a crucial first step in developing a reliable movie recommender platform

This project focuses on building a **Movie Recommender Database** that serves as the foundation for personalized suggestion systems. The database design incorporates essential components such as Users, Movies, Genres, Ratings, Reviews, and Watch History. Each element has been normalized to reduce redundancy and ensure efficient data retrieval. Furthermore, the system includes advanced SQL features such as stored procedures, triggers, indexes, and constraints. These features automate essential activities, maintain data integrity, and enhance the overall performance of the system.

Important procedures like automatic rating insertion, user verification, retrieving top-rated movies, and generating genre-based recommendations help streamline the backend processing. Triggers ensure consistency by preventing duplicate reviews or updating timestamps automatically. The database also supports a recommendation caching mechanism that speeds up results for frequently requested movie lists.

PROBLEM DEFINITION

A modern movie recommendation platform must handle large volumes of data generated by users and the movies they interact with. The core problem lies in designing a database that can effectively manage this data while supporting fast access, personalization, and intelligent suggestions. Without a well-structured database, the entire recommendation process becomes slow, inaccurate, and unreliable.

The first problem addressed in this project is the organization of movie information. Movies contain multiple attributes — titles, genres, release dates, casts, popularity metrics, and descriptions. A database must store this data in a normalized form to avoid duplication and maintain consistency. Improper structuring can lead to redundant records and difficult maintenance. Therefore, designing appropriate tables and relationships becomes essential.

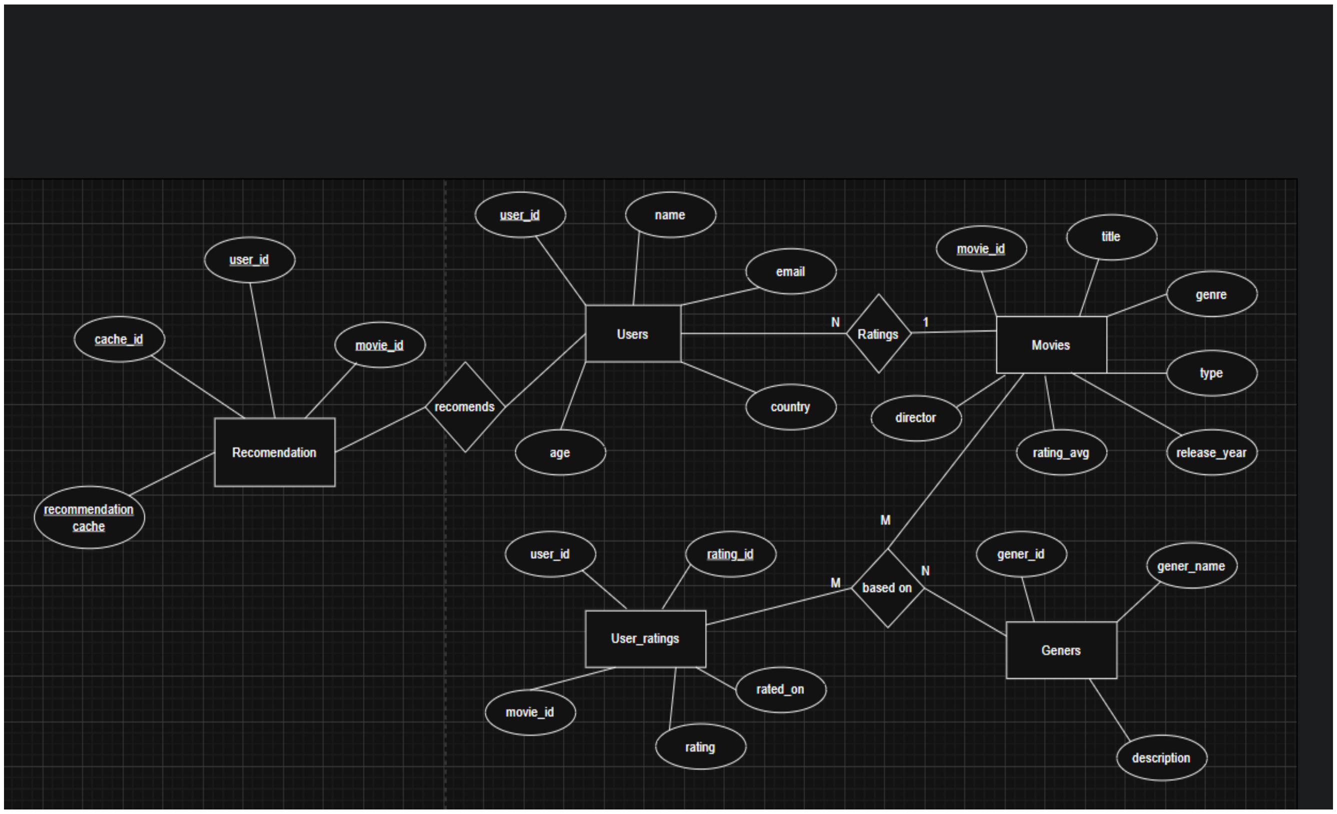
The second problem is the management of user data, including authentication, personal profiles, and email verification. Since users frequently interact with the system, the database must ensure secure storage of credentials, allow quick retrieval of profiles, and support features like unique usernames or email-based login. Ensuring data privacy and validation adds another layer of complexity.

Another critical challenge is handling **user interactions** such as ratings and reviews. Users may rate multiple movies, update their ratings, or provide written feedback. The database must prevent issues like duplicate entries, inconsistent values, or invalid ratings. Triggers and constraints must ensure that only valid actions are allowed. Additionally, the system must maintain a record of user watch history, which plays a major role in generating future recommendations.

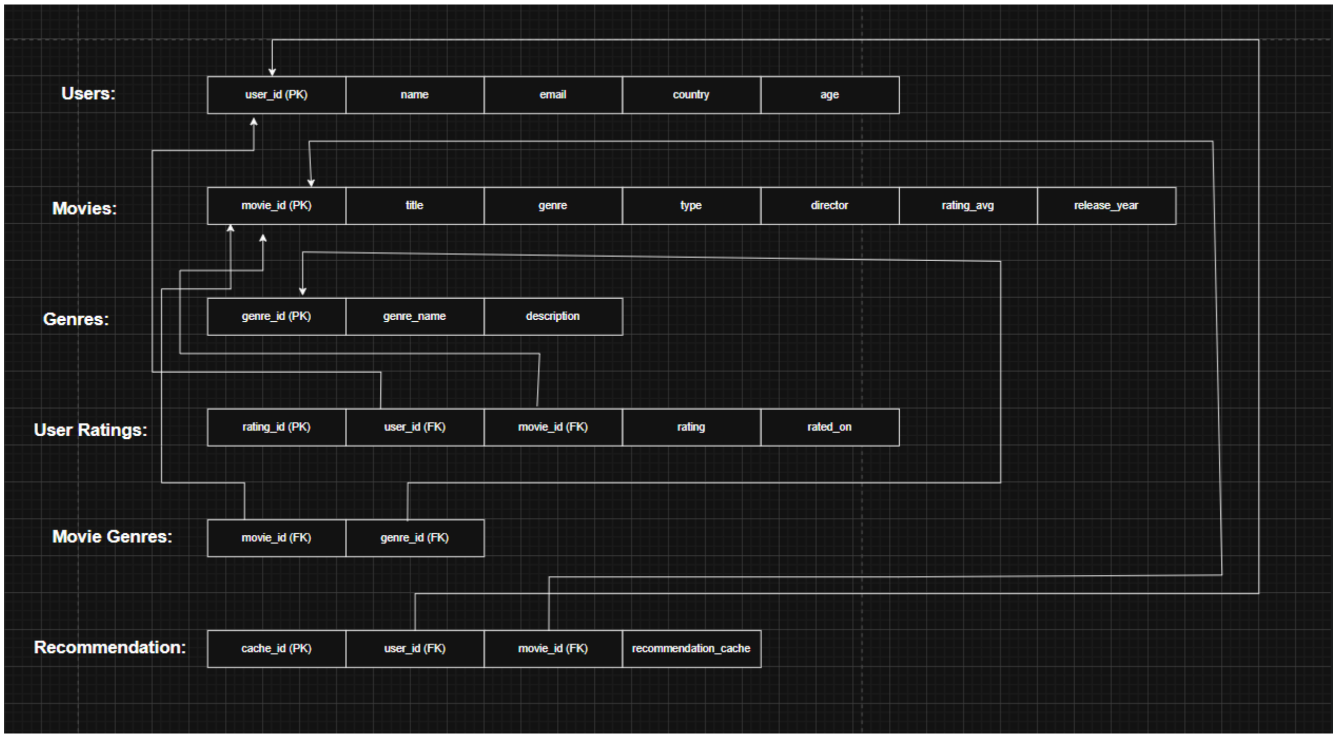
A major problem for recommendation engines is **generating personalized suggestions**. This requires analyzing user preferences, viewing patterns, and frequently watched genres. The database must support quick filtering of movies, retrieving high-rated films, and identifying genre-based suggestions. Stored procedures play an important role here by automating these tasks and reducing processing time.

Finally, the system must address **performance and scalability**. As data grows, queries may slow down, causing delays in recommendations. To solve this, the database includes indexes, optimized joins, and recommendation caching mechanisms. These features ensure that data retrieval remains fast even when dealing with large datasets.

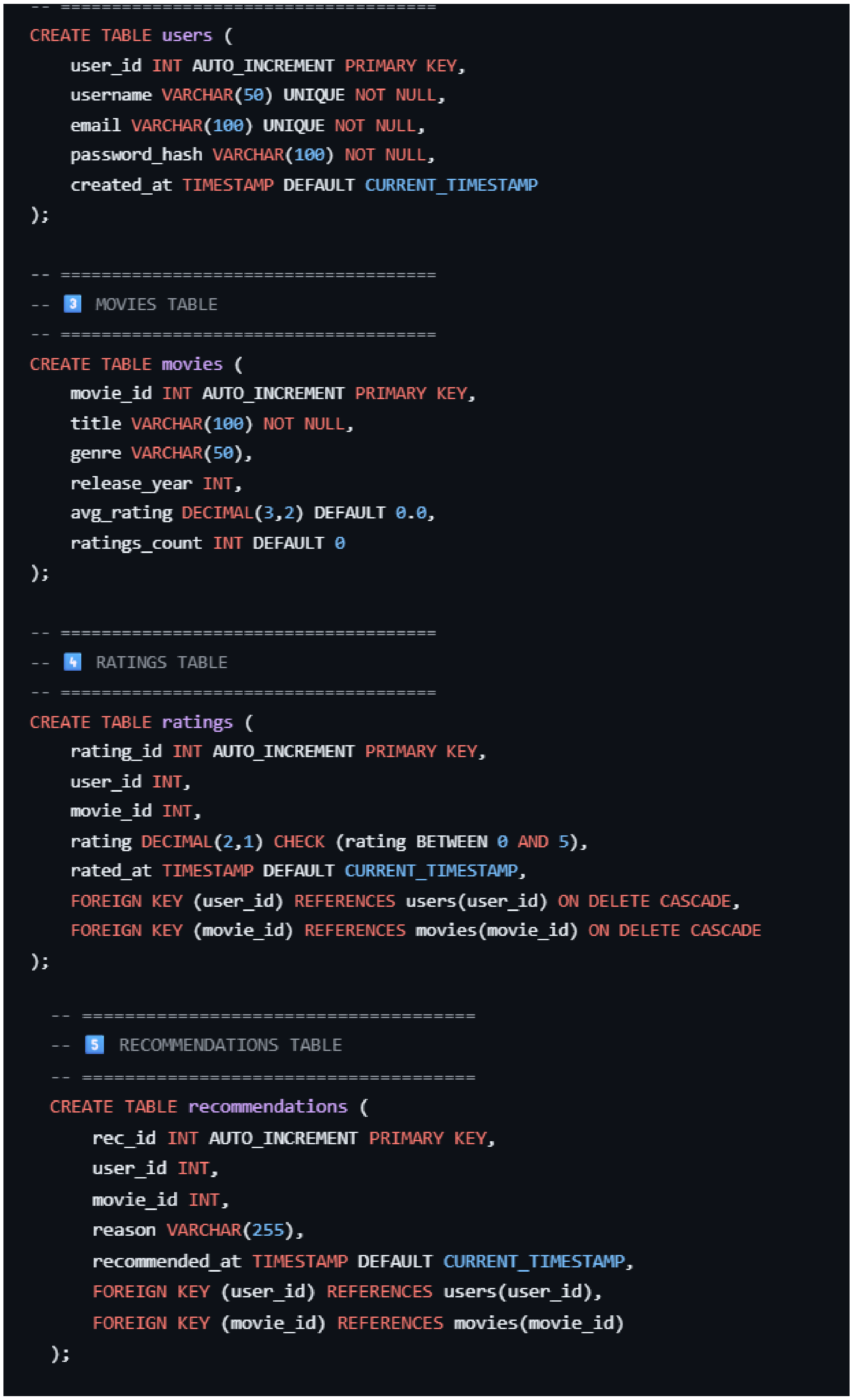
ER MODEL


user_id
name
email
movie_id
title
genre
type
release_year
rating_avg
director
country
age
Users
Ratings
N
1
Movies
cache_id
user_id
movie_id
recomends
Recomendation
recommendation cache
user_id
rating_id
User_ratings
movie_id
rating
rated_on
M
based on
M
N
gener_id
gener_name
Geners
description

ER TO RELATIONAL MAPPING

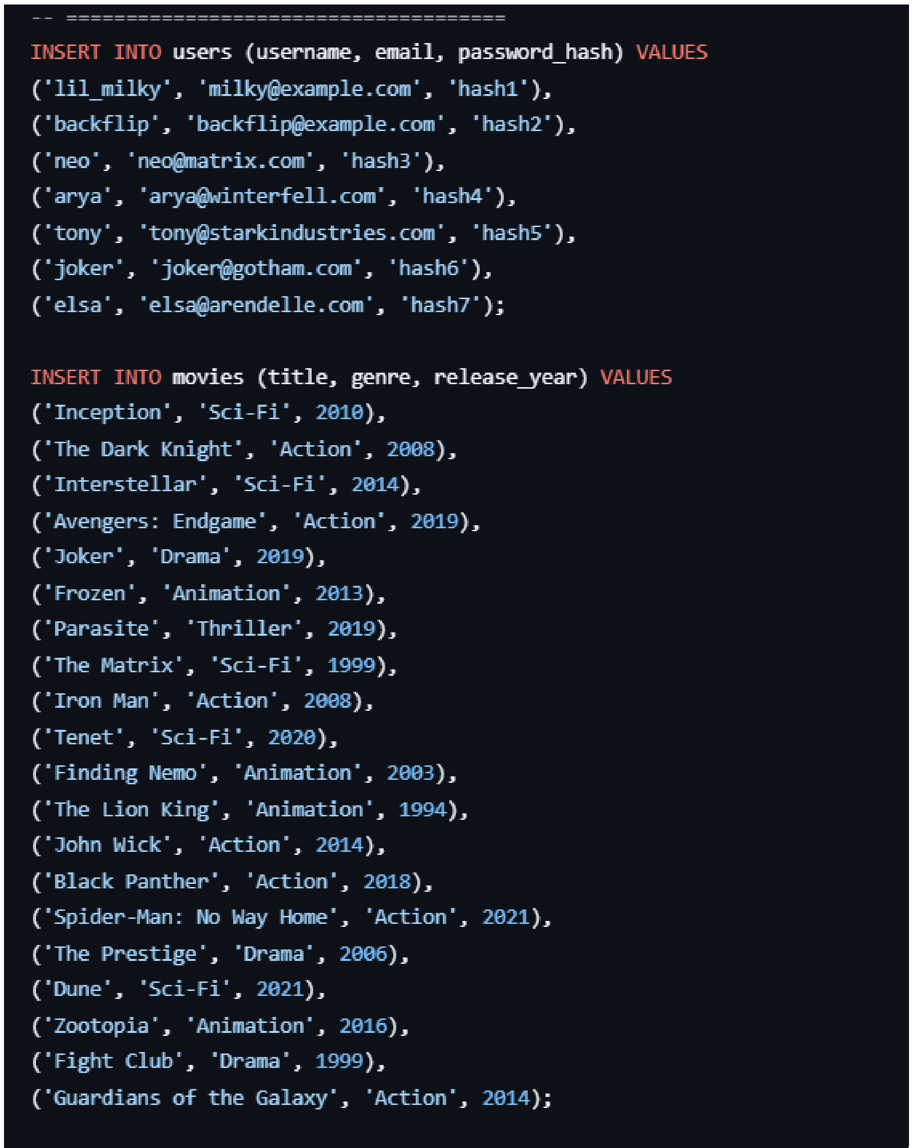


DDL STATEMENTS

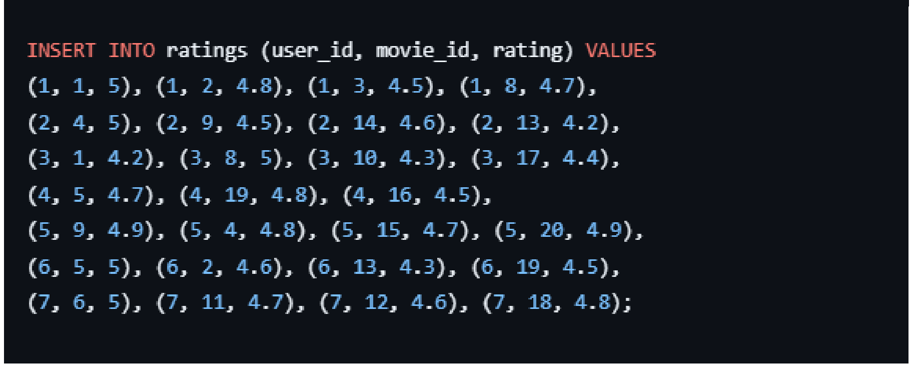


```sql
-- =====================================
CREATE TABLE users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password_hash VARCHAR(100) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- =====================================
-- 3️⃣ MOVIES TABLE
-- =====================================
CREATE TABLE movies (
    movie_id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(100) NOT NULL,
    genre VARCHAR(50),
    release_year INT,
    avg_rating DECIMAL(3,2) DEFAULT 0.0,
    ratings_count INT DEFAULT 0
);

-- =====================================
-- 4️⃣ RATINGS TABLE
-- =====================================
CREATE TABLE ratings (
    rating_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    movie_id INT,
    rating DECIMAL(2,1) CHECK (rating BETWEEN 0 AND 5),
    rated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
    FOREIGN KEY (movie_id) REFERENCES movies(movie_id) ON DELETE CASCADE
);

  -- =====================================
  -- 5️⃣ RECOMMENDATIONS TABLE
  -- =====================================
  CREATE TABLE recommendations (
      rec_id INT AUTO_INCREMENT PRIMARY KEY,
      user_id INT,
      movie_id INT,
      reason VARCHAR(255),
      recommended_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
      FOREIGN KEY (user_id) REFERENCES users(user_id),
      FOREIGN KEY (movie_id) REFERENCES movies(movie_id)
  );
```

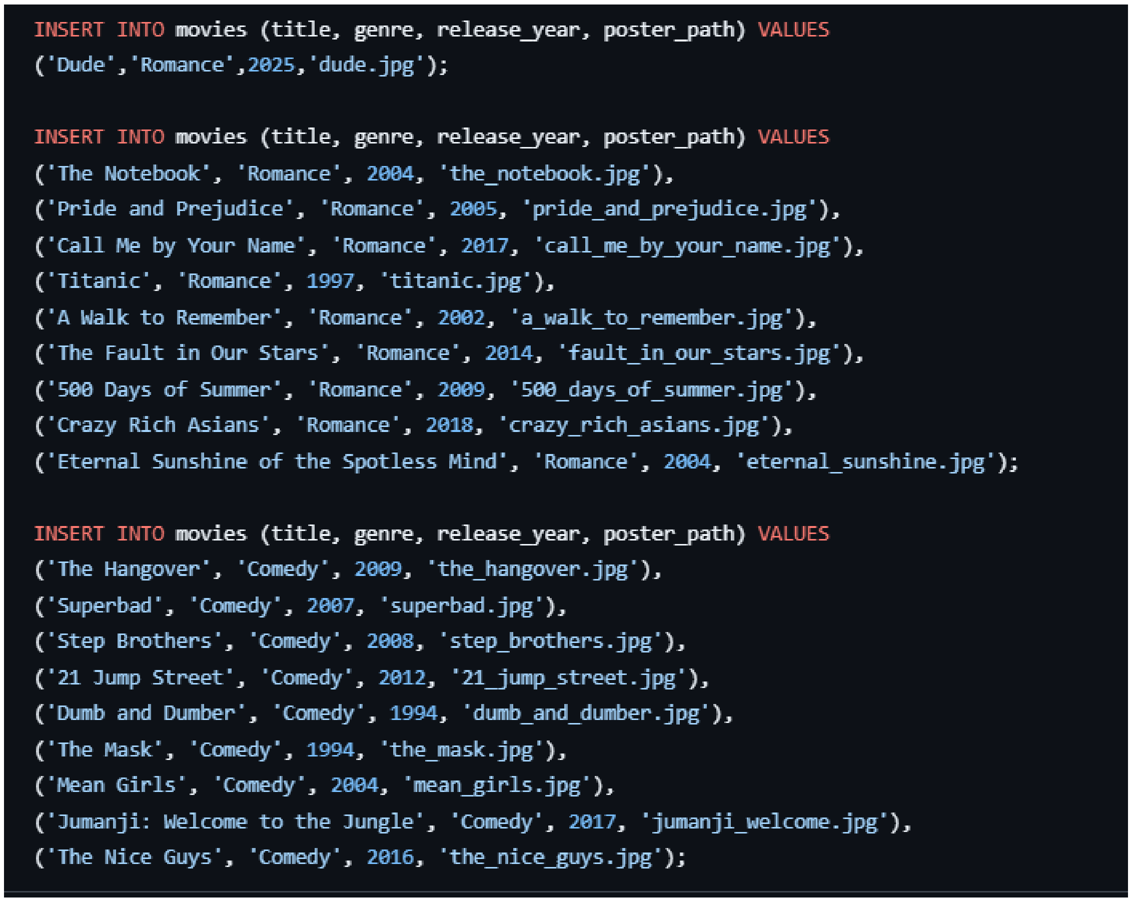
DML STATEMENTS



```sql
-- =====================================
INSERT INTO users (username, email, password_hash) VALUES
('lil_milky', 'milky@example.com', 'hash1'),
('backflip', 'backflip@example.com', 'hash2'),
('neo', 'neo@matrix.com', 'hash3'),
('arya', 'arya@winterfell.com', 'hash4'),
('tony', 'tony@starkindustries.com', 'hash5'),
('joker', 'joker@gotham.com', 'hash6'),
('elsa', 'elsa@arendelle.com', 'hash7');

INSERT INTO movies (title, genre, release_year) VALUES
('Inception', 'Sci-Fi', 2010),
('The Dark Knight', 'Action', 2008),
('Interstellar', 'Sci-Fi', 2014),
('Avengers: Endgame', 'Action', 2019),
('Joker', 'Drama', 2019),
('Frozen', 'Animation', 2013),
('Parasite', 'Thriller', 2019),
('The Matrix', 'Sci-Fi', 1999),
('Iron Man', 'Action', 2008),
('Tenet', 'Sci-Fi', 2020),
('Finding Nemo', 'Animation', 2003),
('The Lion King', 'Animation', 1994),
('John Wick', 'Action', 2014),
('Black Panther', 'Action', 2018),
('Spider-Man: No Way Home', 'Action', 2021),
('The Prestige', 'Drama', 2006),
('Dune', 'Sci-Fi', 2021),
('Zootopia', 'Animation', 2016),
('Fight Club', 'Drama', 1999),
('Guardians of the Galaxy', 'Action', 2014);
```

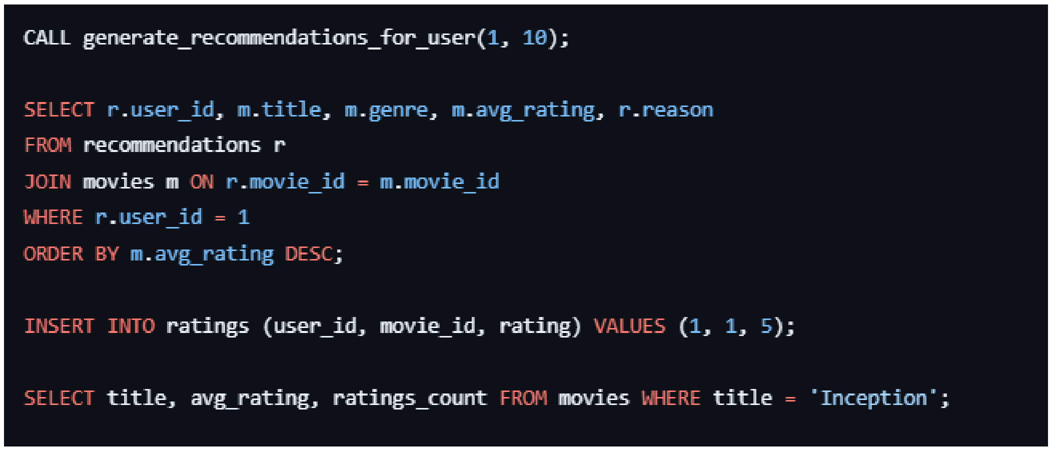


```sql
INSERT INTO ratings (user_id, movie_id, rating) VALUES
(1, 1, 5), (1, 2, 4.8), (1, 3, 4.5), (1, 8, 4.7),
(2, 4, 5), (2, 9, 4.5), (2, 14, 4.6), (2, 13, 4.2),
(3, 1, 4.2), (3, 8, 5), (3, 10, 4.3), (3, 17, 4.4),
(4, 5, 4.7), (4, 19, 4.8), (4, 16, 4.5),
(5, 9, 4.9), (5, 4, 4.8), (5, 15, 4.7), (5, 20, 4.9),
(6, 5, 5), (6, 2, 4.6), (6, 13, 4.3), (6, 19, 4.5),
(7, 6, 5), (7, 11, 4.7), (7, 12, 4.6), (7, 18, 4.8);
```

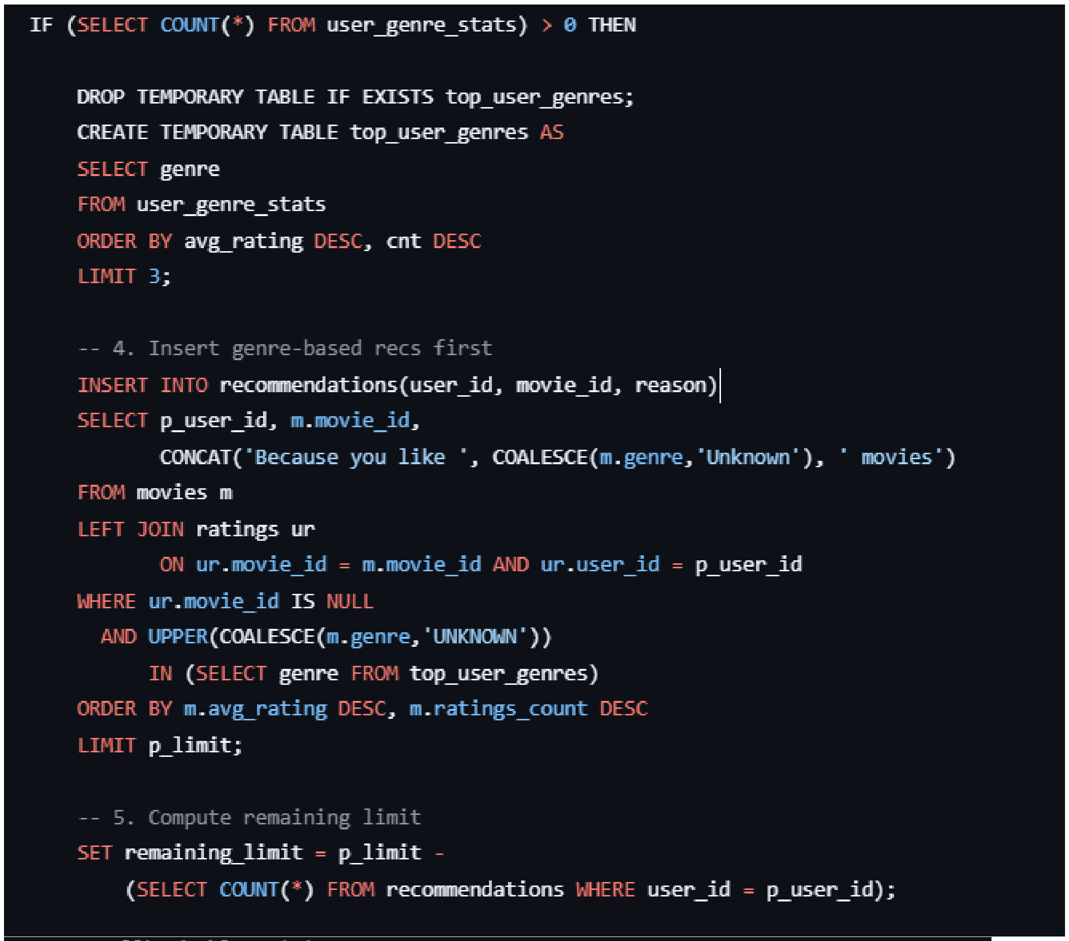


```sql
INSERT INTO movies (title, genre, release_year, poster_path) VALUES
('Dude','Romance',2025,'dude.jpg');

INSERT INTO movies (title, genre, release_year, poster_path) VALUES
('The Notebook', 'Romance', 2004, 'the_notebook.jpg'),
('Pride and Prejudice', 'Romance', 2005, 'pride_and_prejudice.jpg'),
('Call Me by Your Name', 'Romance', 2017, 'call_me_by_your_name.jpg'),
('Titanic', 'Romance', 1997, 'titanic.jpg'),
('A Walk to Remember', 'Romance', 2002, 'a_walk_to_remember.jpg'),
('The Fault in Our Stars', 'Romance', 2014, 'fault_in_our_stars.jpg'),
('500 Days of Summer', 'Romance', 2009, '500_days_of_summer.jpg'),
('Crazy Rich Asians', 'Romance', 2018, 'crazy_rich_asians.jpg'),
('Eternal Sunshine of the Spotless Mind', 'Romance', 2004, 'eternal_sunshine.jpg');

INSERT INTO movies (title, genre, release_year, poster_path) VALUES
('The Hangover', 'Comedy', 2009, 'the_hangover.jpg'),
('Superbad', 'Comedy', 2007, 'superbad.jpg'),
('Step Brothers', 'Comedy', 2008, 'step_brothers.jpg'),
('21 Jump Street', 'Comedy', 2012, '21_jump_street.jpg'),
('Dumb and Dumber', 'Comedy', 1994, 'dumb_and_dumber.jpg'),
('The Mask', 'Comedy', 1994, 'the_mask.jpg'),
('Mean Girls', 'Comedy', 2004, 'mean_girls.jpg'),
('Jumanji: Welcome to the Jungle', 'Comedy', 2017, 'jumanji_welcome.jpg'),
('The Nice Guys', 'Comedy', 2016, 'the_nice_guys.jpg');
```

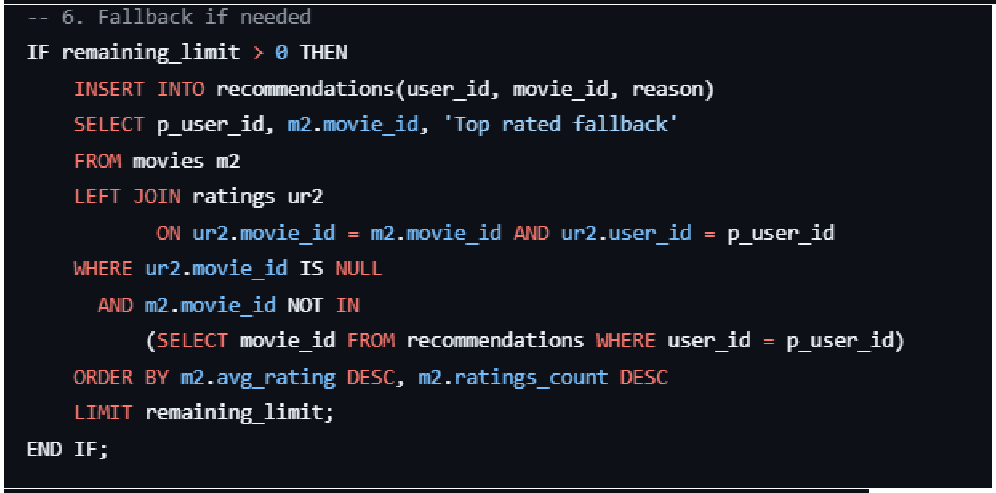
QUERIES



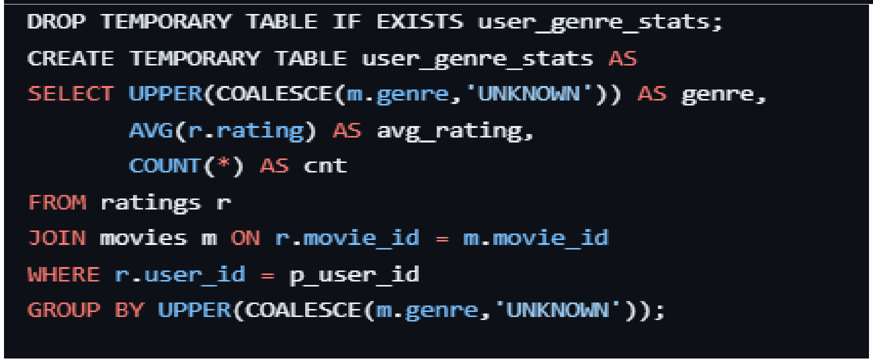
```sql
CALL generate_recommendations_for_user(1, 10);

SELECT r.user_id, m.title, m.genre, m.avg_rating, r.reason
FROM recommendations r
JOIN movies m ON r.movie_id = m.movie_id
WHERE r.user_id = 1
ORDER BY m.avg_rating DESC;

INSERT INTO ratings (user_id, movie_id, rating) VALUES (1, 1, 5);

SELECT title, avg_rating, ratings_count FROM movies WHERE title = 'Inception';
```



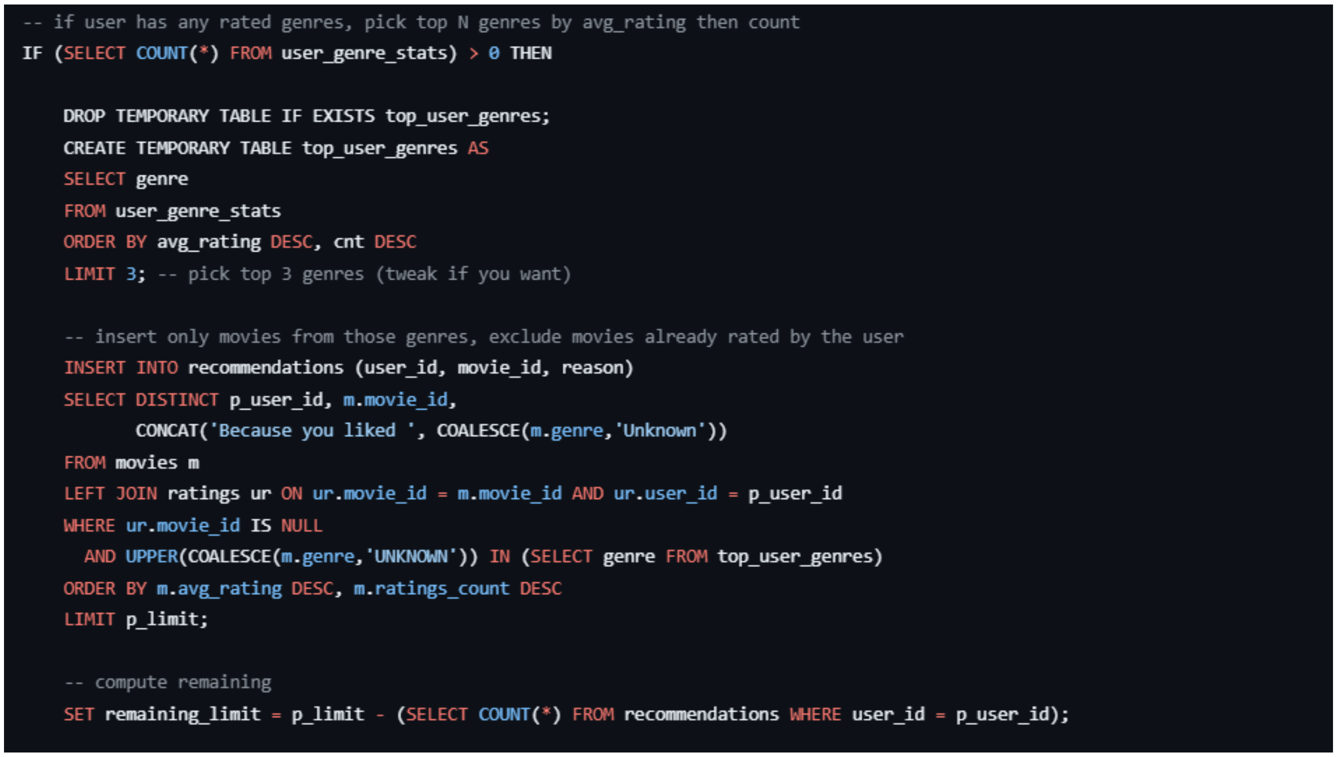
```sql
IF (SELECT COUNT(*) FROM user_genre_stats) > 0 THEN

    DROP TEMPORARY TABLE IF EXISTS top_user_genres;
    CREATE TEMPORARY TABLE top_user_genres AS
    SELECT genre
    FROM user_genre_stats
    ORDER BY avg_rating DESC, cnt DESC
    LIMIT 3;

    -- 4. Insert genre-based recs first
    INSERT INTO recommendations(user_id, movie_id, reason)
    SELECT p_user_id, m.movie_id,
           CONCAT('Because you like ', COALESCE(m.genre,'Unknown'), ' movies')
    FROM movies m
    LEFT JOIN ratings ur
           ON ur.movie_id = m.movie_id AND ur.user_id = p_user_id
    WHERE ur.movie_id IS NULL
      AND UPPER(COALESCE(m.genre,'UNKNOWN'))
          IN (SELECT genre FROM top_user_genres)
    ORDER BY m.avg_rating DESC, m.ratings_count DESC
    LIMIT p_limit;

    -- 5. Compute remaining limit
    SET remaining_limit = p_limit -
        (SELECT COUNT(*) FROM recommendations WHERE user_id = p_user_id);
```



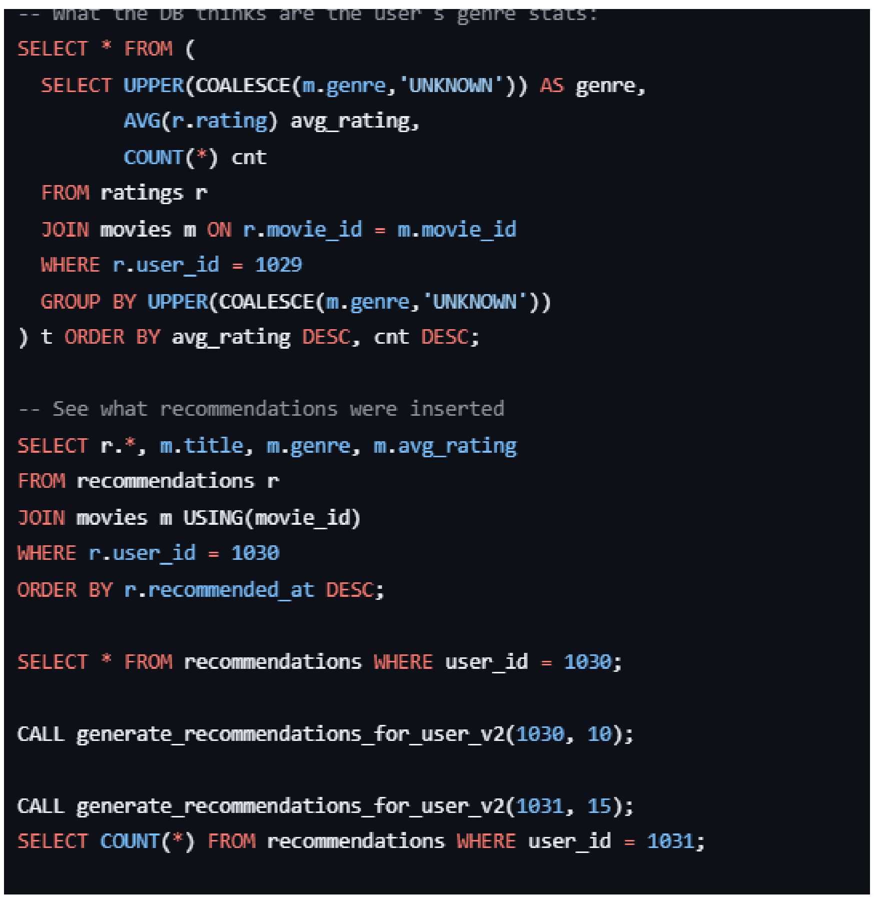
```sql
-- 6. Fallback if needed
IF remaining_limit > 0 THEN
    INSERT INTO recommendations(user_id, movie_id, reason)
    SELECT p_user_id, m2.movie_id, 'Top rated fallback'
    FROM movies m2
    LEFT JOIN ratings ur2
           ON ur2.movie_id = m2.movie_id AND ur2.user_id = p_user_id
    WHERE ur2.movie_id IS NULL
      AND m2.movie_id NOT IN
          (SELECT movie_id FROM recommendations WHERE user_id = p_user_id)
    ORDER BY m2.avg_rating DESC, m2.ratings_count DESC
    LIMIT remaining_limit;
END IF;
```



```sql
DROP TEMPORARY TABLE IF EXISTS user_genre_stats;
CREATE TEMPORARY TABLE user_genre_stats AS
SELECT UPPER(COALESCE(m.genre,'UNKNOWN')) AS genre,
       AVG(r.rating) AS avg_rating,
       COUNT(*) AS cnt
FROM ratings r
JOIN movies m ON r.movie_id = m.movie_id
WHERE r.user_id = p_user_id
GROUP BY UPPER(COALESCE(m.genre,'UNKNOWN'));
```



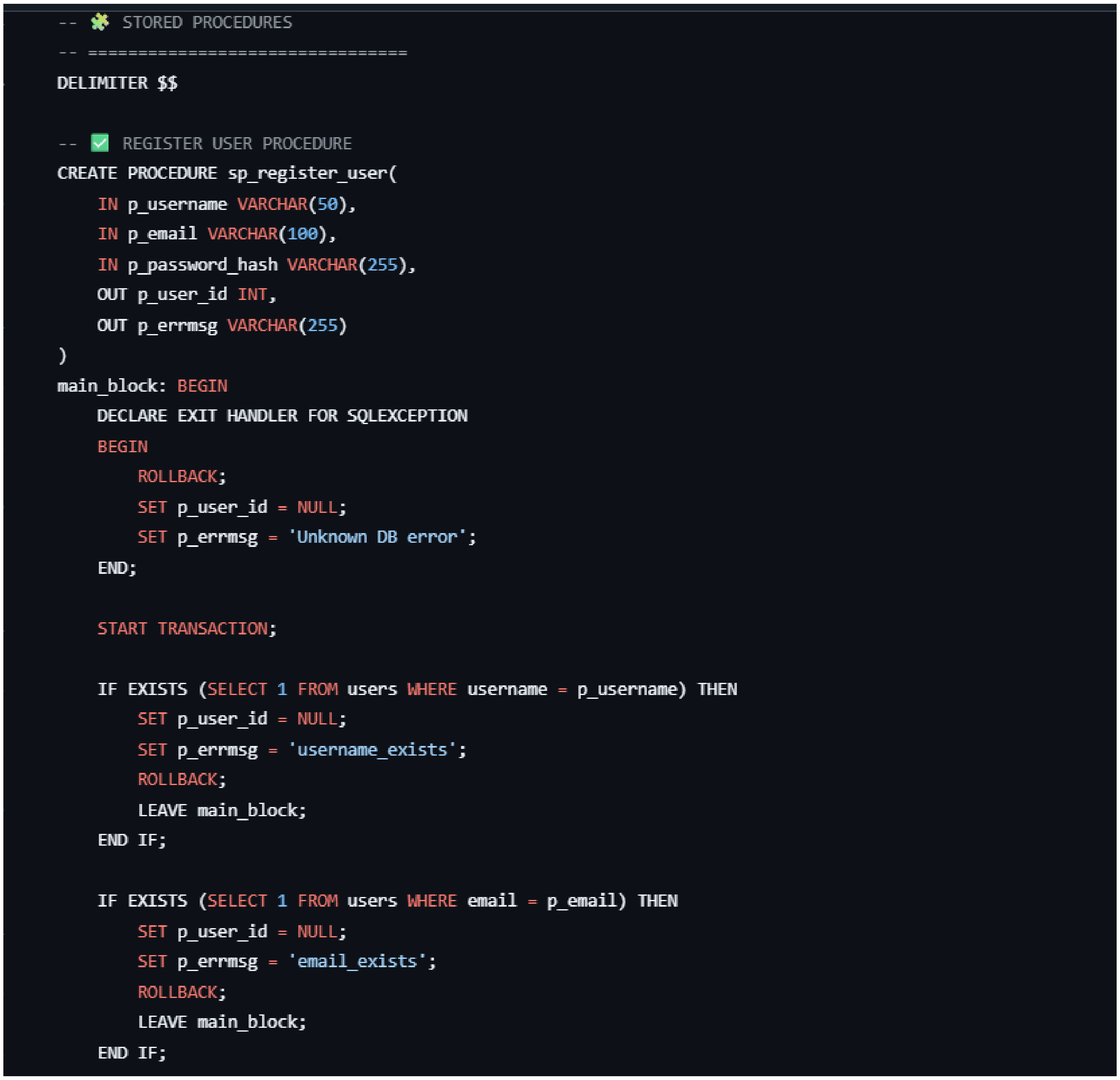
```sql
-- if user has any rated genres, pick top N genres by avg_rating then count
IF (SELECT COUNT(*) FROM user_genre_stats) > 0 THEN

    DROP TEMPORARY TABLE IF EXISTS top_user_genres;
    CREATE TEMPORARY TABLE top_user_genres AS
    SELECT genre
    FROM user_genre_stats
    ORDER BY avg_rating DESC, cnt DESC
    LIMIT 3; -- pick top 3 genres (tweak if you want)

    -- insert only movies from those genres, exclude movies already rated by the user
    INSERT INTO recommendations (user_id, movie_id, reason)
    SELECT DISTINCT p_user_id, m.movie_id,
           CONCAT('Because you liked ', COALESCE(m.genre,'Unknown'))
    FROM movies m
    LEFT JOIN ratings ur ON ur.movie_id = m.movie_id AND ur.user_id = p_user_id
    WHERE ur.movie_id IS NULL
      AND UPPER(COALESCE(m.genre,'UNKNOWN')) IN (SELECT genre FROM top_user_genres)
    ORDER BY m.avg_rating DESC, m.ratings_count DESC
    LIMIT p_limit;

    -- compute remaining
    SET remaining_limit = p_limit - (SELECT COUNT(*) FROM recommendations WHERE user_id = p_user_id);
```



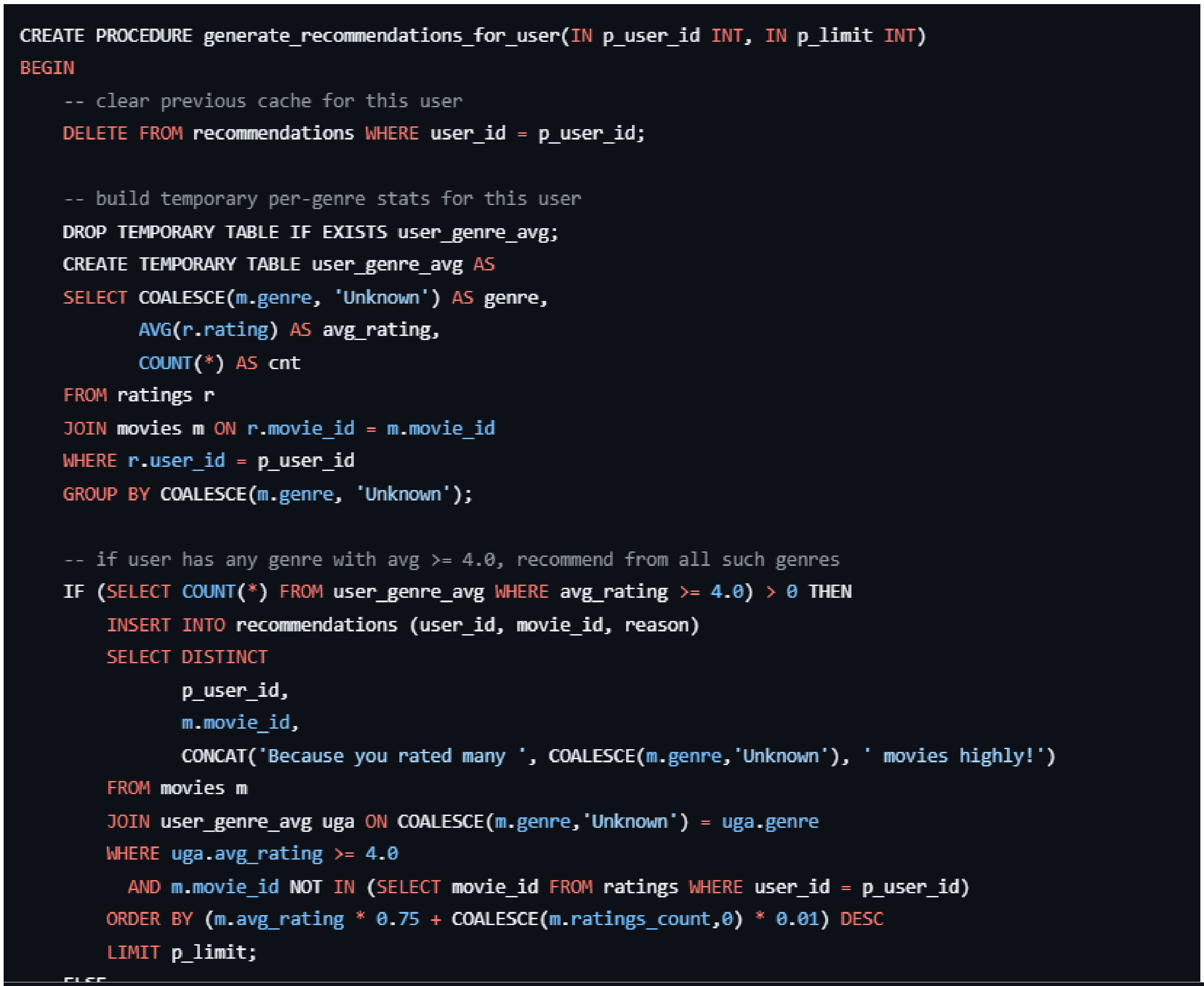
```sql
-- What the DB thinks are the user's genre stats?
SELECT * FROM (
  SELECT UPPER(COALESCE(m.genre,'UNKNOWN')) AS genre,
         AVG(r.rating) avg_rating,
         COUNT(*) cnt
  FROM ratings r
  JOIN movies m ON r.movie_id = m.movie_id
  WHERE r.user_id = 1029
  GROUP BY UPPER(COALESCE(m.genre,'UNKNOWN'))
) t ORDER BY avg_rating DESC, cnt DESC;

-- See what recommendations were inserted
SELECT r.*, m.title, m.genre, m.avg_rating
FROM recommendations r
JOIN movies m USING(movie_id)
WHERE r.user_id = 1030
ORDER BY r.recommended_at DESC;

SELECT * FROM recommendations WHERE user_id = 1030;

CALL generate_recommendations_for_user_v2(1030, 10);

CALL generate_recommendations_for_user_v2(1031, 15);
SELECT COUNT(*) FROM recommendations WHERE user_id = 1031;
```

STORED PROCEDURES, FUCNTIONS AND TRIGGERS

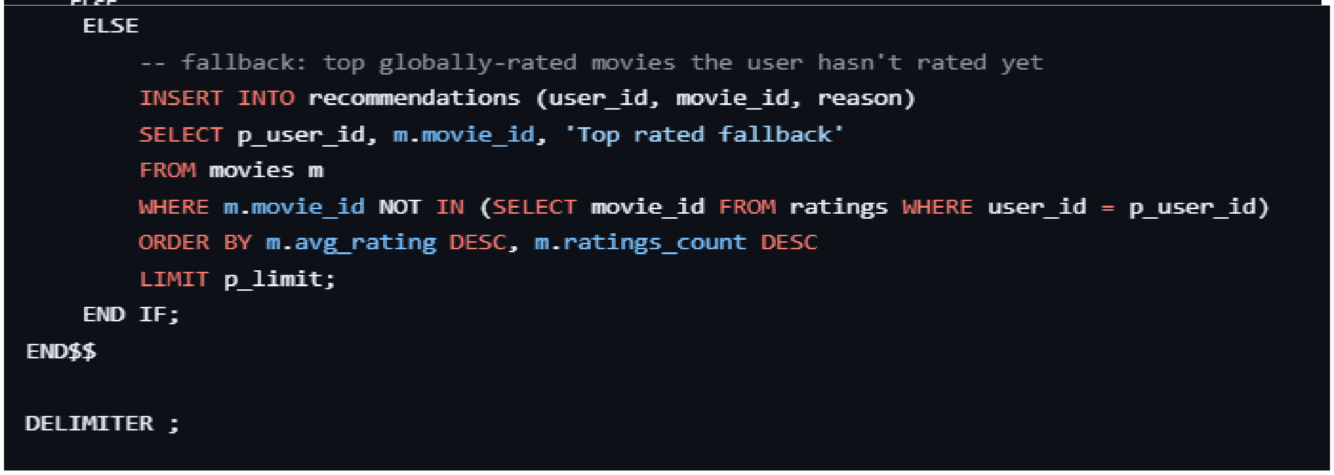
1.STORED PROCEDURES OR FUNCTIONS



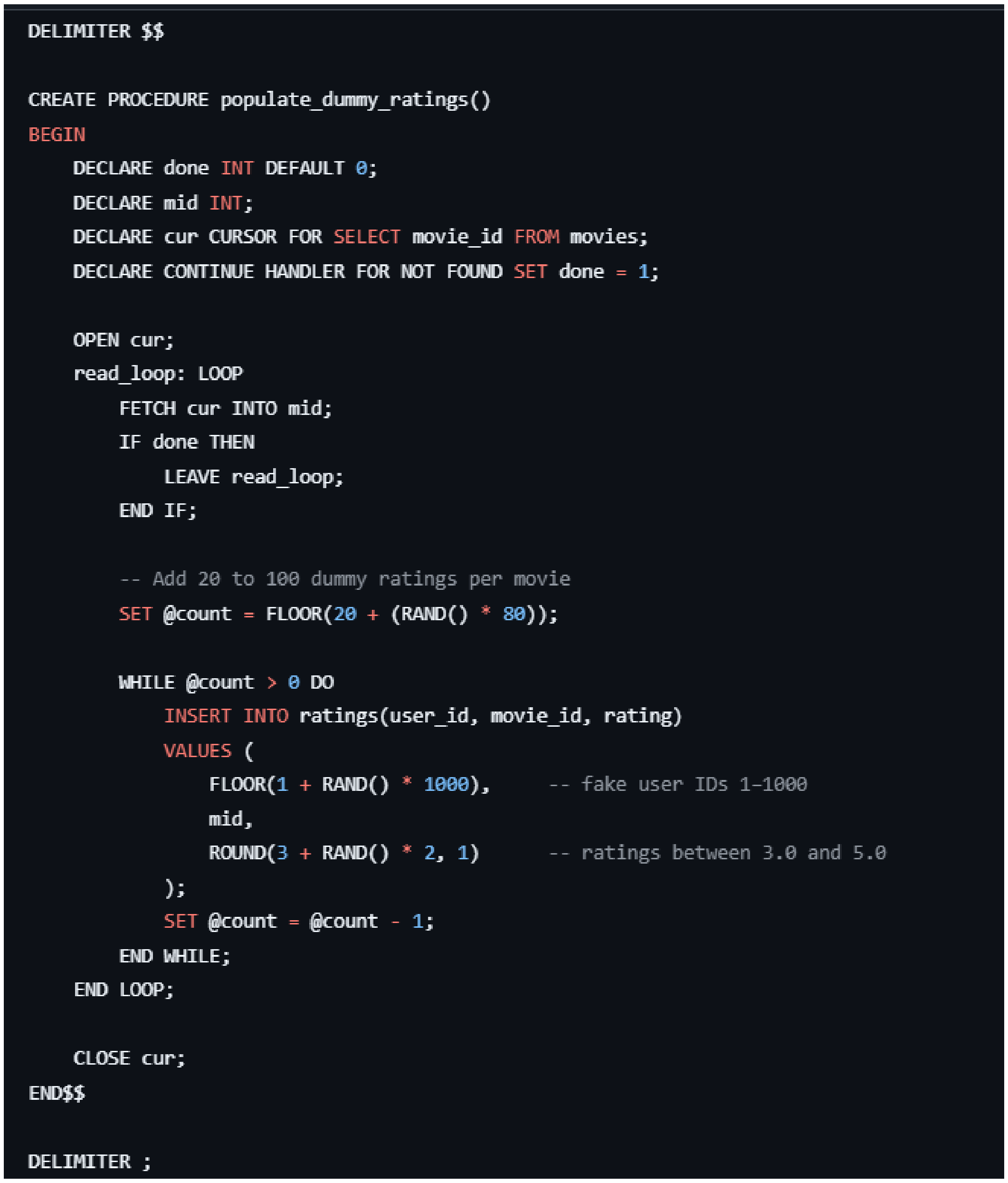
```sql
-- 🧩 STORED PROCEDURES
-- ================================
DELIMITER $$

-- ✅ REGISTER USER PROCEDURE
CREATE PROCEDURE sp_register_user(
    IN p_username VARCHAR(50),
    IN p_email VARCHAR(100),
    IN p_password_hash VARCHAR(255),
    OUT p_user_id INT,
    OUT p_errmsg VARCHAR(255)
)
main_block: BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        SET p_user_id = NULL;
        SET p_errmsg = 'Unknown DB error';
    END;

    START TRANSACTION;

    IF EXISTS (SELECT 1 FROM users WHERE username = p_username) THEN
        SET p_user_id = NULL;
        SET p_errmsg = 'username_exists';
        ROLLBACK;
        LEAVE main_block;
    END IF;

    IF EXISTS (SELECT 1 FROM users WHERE email = p_email) THEN
        SET p_user_id = NULL;
        SET p_errmsg = 'email_exists';
        ROLLBACK;
        LEAVE main_block;
    END IF;
```



```sql
CREATE PROCEDURE generate_recommendations_for_user(IN p_user_id INT, IN p_limit INT)
BEGIN
    -- clear previous cache for this user
    DELETE FROM recommendations WHERE user_id = p_user_id;

    -- build temporary per-genre stats for this user
    DROP TEMPORARY TABLE IF EXISTS user_genre_avg;
    CREATE TEMPORARY TABLE user_genre_avg AS
    SELECT COALESCE(m.genre, 'Unknown') AS genre,
           AVG(r.rating) AS avg_rating,
           COUNT(*) AS cnt
    FROM ratings r
    JOIN movies m ON r.movie_id = m.movie_id
    WHERE r.user_id = p_user_id
    GROUP BY COALESCE(m.genre, 'Unknown');

    -- if user has any genre with avg >= 4.0, recommend from all such genres
    IF (SELECT COUNT(*) FROM user_genre_avg WHERE avg_rating >= 4.0) > 0 THEN
        INSERT INTO recommendations (user_id, movie_id, reason)
        SELECT DISTINCT
               p_user_id,
               m.movie_id,
               CONCAT('Because you rated many ', COALESCE(m.genre,'Unknown'), ' movies highly!')
        FROM movies m
        JOIN user_genre_avg uga ON COALESCE(m.genre,'Unknown') = uga.genre
        WHERE uga.avg_rating >= 4.0
          AND m.movie_id NOT IN (SELECT movie_id FROM ratings WHERE user_id = p_user_id)
        ORDER BY (m.avg_rating * 0.75 + COALESCE(m.ratings_count,0) * 0.01) DESC
        LIMIT p_limit;
```

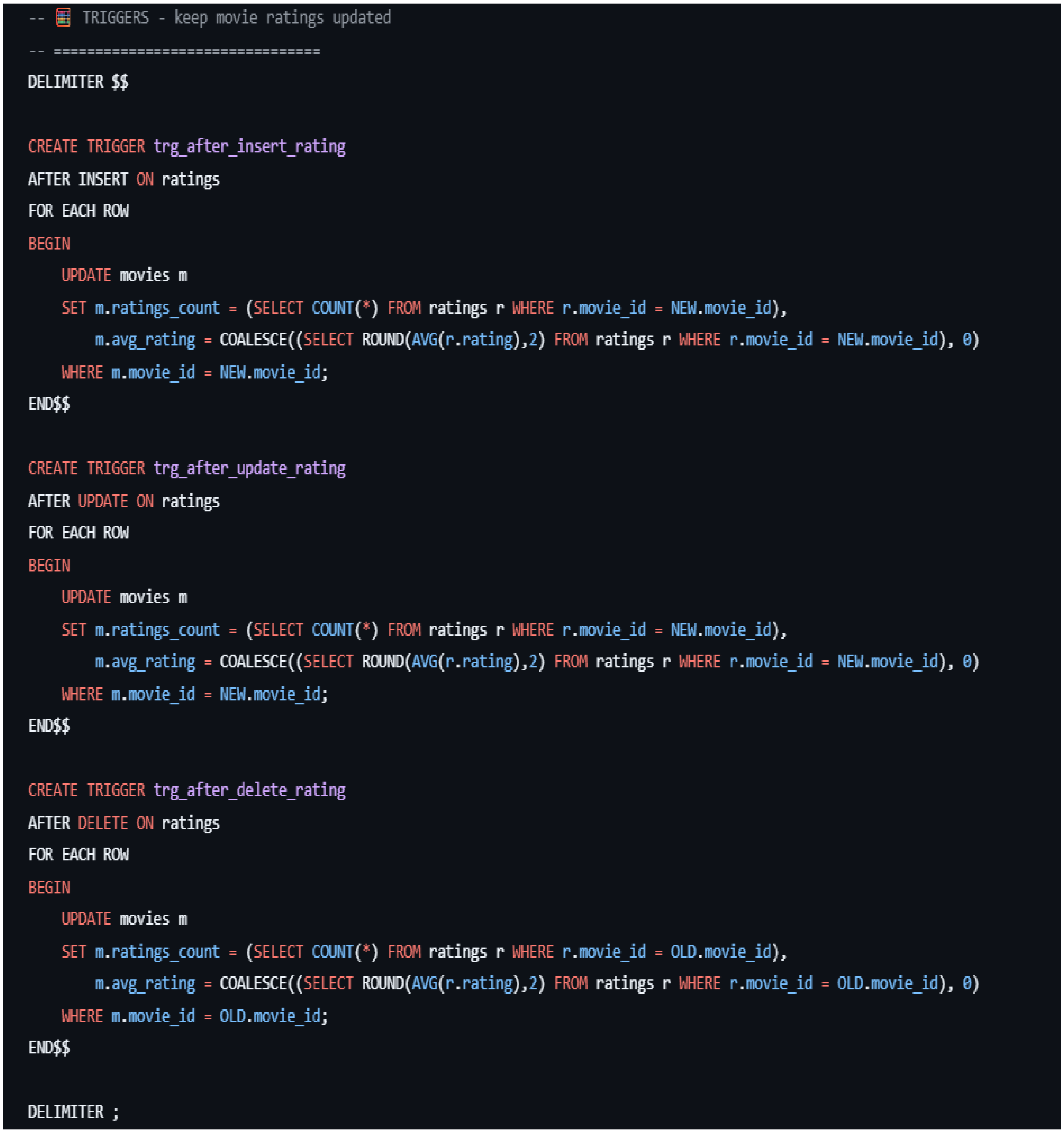


```sql
    ELSE
        -- fallback: top globally-rated movies the user hasn't rated yet
        INSERT INTO recommendations (user_id, movie_id, reason)
        SELECT p_user_id, m.movie_id, 'Top rated fallback'
        FROM movies m
        WHERE m.movie_id NOT IN (SELECT movie_id FROM ratings WHERE user_id = p_user_id)
        ORDER BY m.avg_rating DESC, m.ratings_count DESC
        LIMIT p_limit;
    END IF;
END$$

DELIMITER ;
```



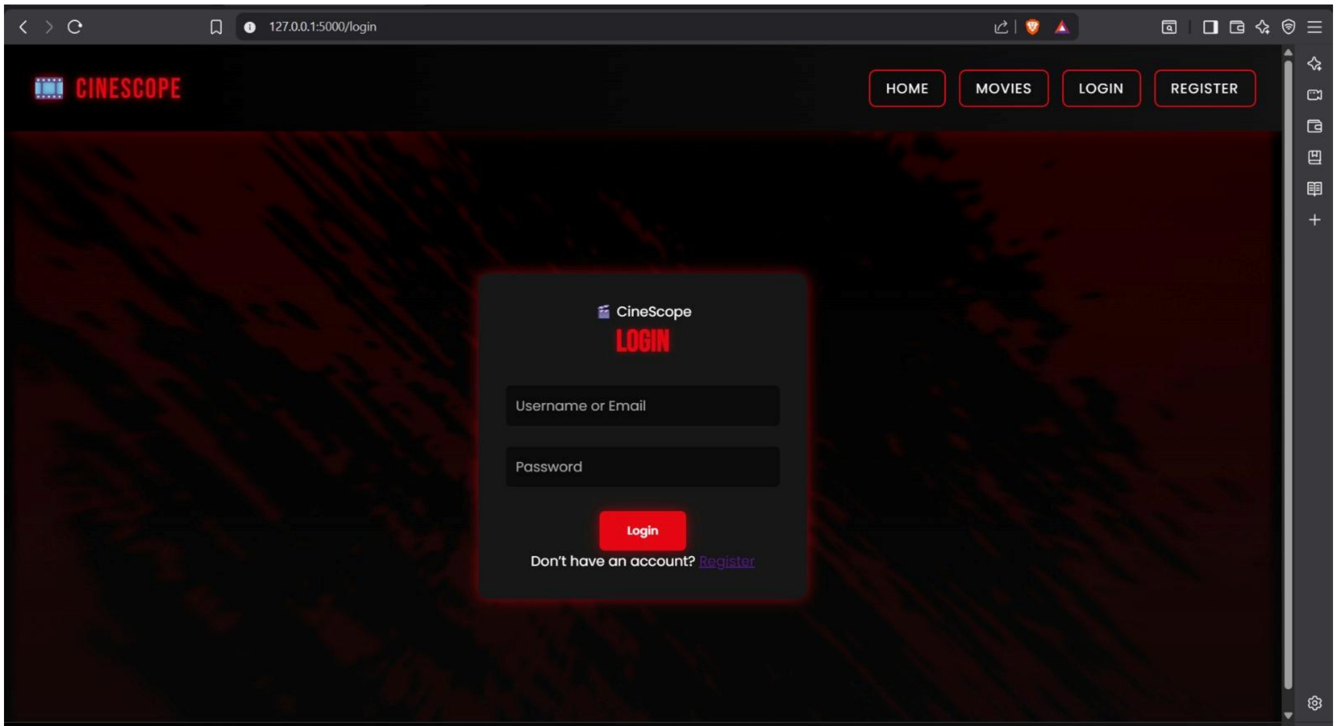
```sql
DELIMITER $$

CREATE PROCEDURE populate_dummy_ratings()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE mid INT;
    DECLARE cur CURSOR FOR SELECT movie_id FROM movies;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN cur;
    read_loop: LOOP
        FETCH cur INTO mid;
        IF done THEN
            LEAVE read_loop;
        END IF;

        -- Add 20 to 100 dummy ratings per movie
        SET @count = FLOOR(20 + (RAND() * 80));

        WHILE @count > 0 DO
            INSERT INTO ratings(user_id, movie_id, rating)
            VALUES (
                FLOOR(1 + RAND() * 1000),     -- fake user IDs 1-1000
                mid,
                ROUND(3 + RAND() * 2, 1)      -- ratings between 3.0 and 5.0
            );
            SET @count = @count - 1;
        END WHILE;
    END LOOP;

    CLOSE cur;
END$$

DELIMITER ;
```

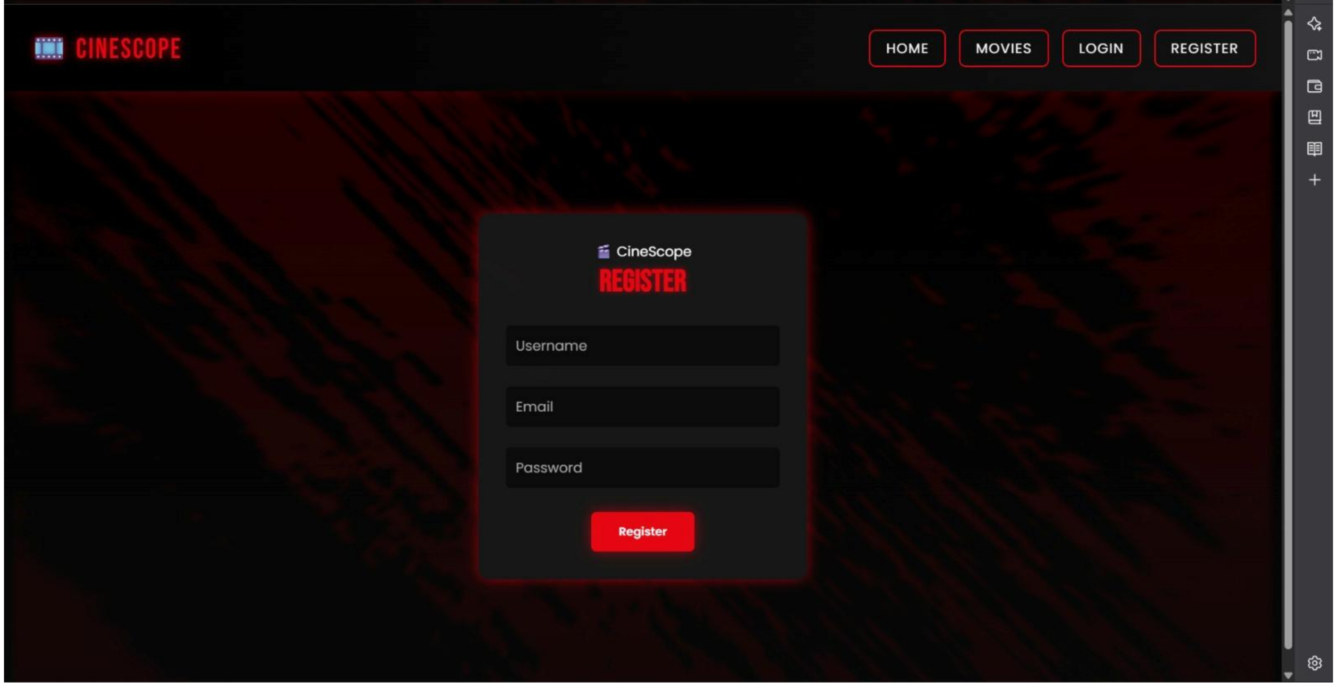
2.TRIGGERS

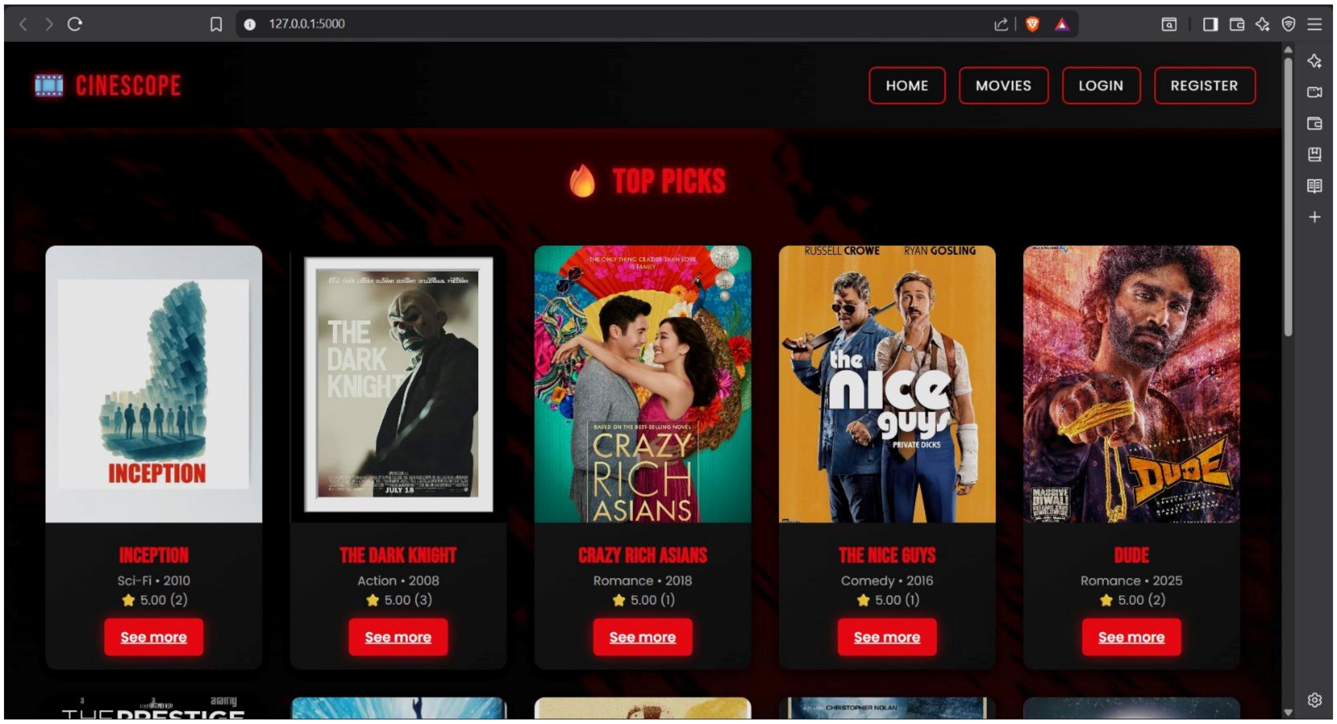


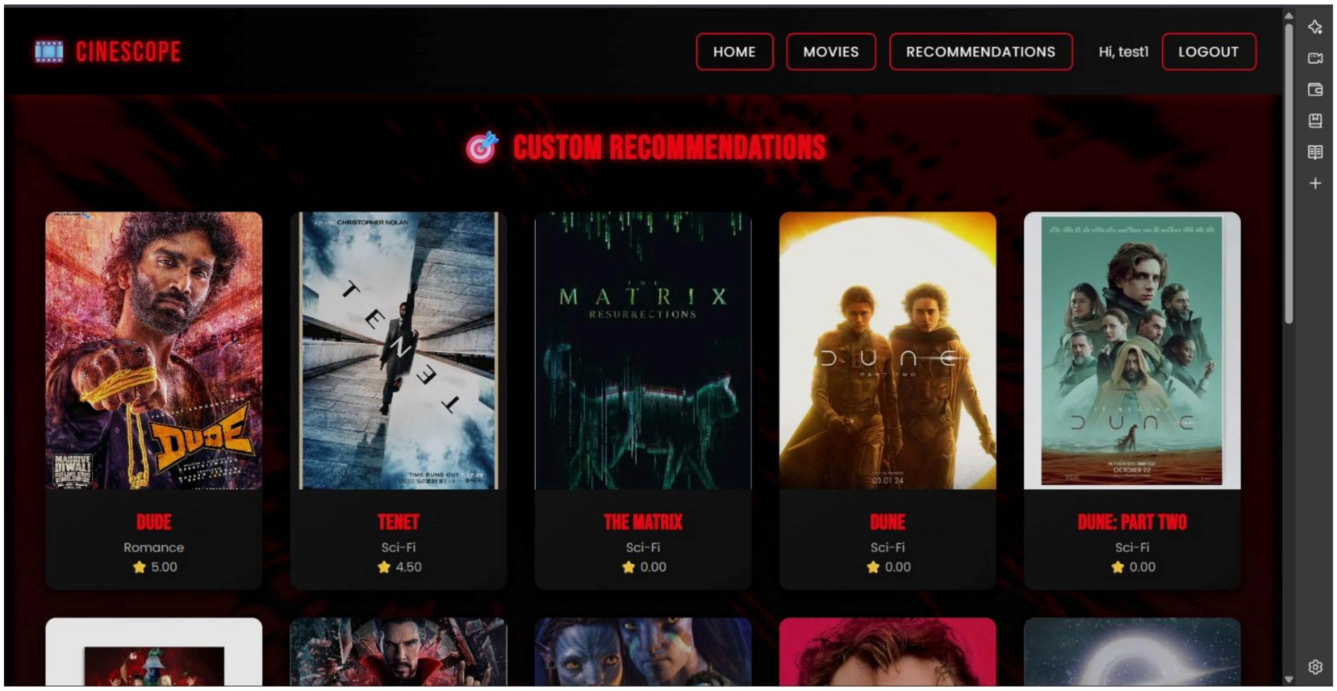
```sql
-- 🧮 TRIGGERS - keep movie ratings updated
-- ================================
DELIMITER $$

CREATE TRIGGER trg_after_insert_rating
AFTER INSERT ON ratings
FOR EACH ROW
BEGIN
    UPDATE movies m
    SET m.ratings_count = (SELECT COUNT(*) FROM ratings r WHERE r.movie_id = NEW.movie_id),
        m.avg_rating = COALESCE((SELECT ROUND(AVG(r.rating),2) FROM ratings r WHERE r.movie_id = NEW.movie_id), 0)
    WHERE m.movie_id = NEW.movie_id;
END$$

CREATE TRIGGER trg_after_update_rating
AFTER UPDATE ON ratings
FOR EACH ROW
BEGIN
    UPDATE movies m
    SET m.ratings_count = (SELECT COUNT(*) FROM ratings r WHERE r.movie_id = NEW.movie_id),
        m.avg_rating = COALESCE((SELECT ROUND(AVG(r.rating),2) FROM ratings r WHERE r.movie_id = NEW.movie_id), 0)
    WHERE m.movie_id = NEW.movie_id;
END$$

CREATE TRIGGER trg_after_delete_rating
AFTER DELETE ON ratings
FOR EACH ROW
BEGIN
    UPDATE movies m
    SET m.ratings_count = (SELECT COUNT(*) FROM ratings r WHERE r.movie_id = OLD.movie_id),
        m.avg_rating = COALESCE((SELECT ROUND(AVG(r.rating),2) FROM ratings r WHERE r.movie_id = OLD.movie_id), 0)
    WHERE m.movie_id = OLD.movie_id;
END$$

DELIMITER ;
```

FRONT END DEVELOPEMNT









GIT hub repo: https://github.com/PES2UG24CS801/Movie_recommedation_DBMS